September 2017

Pipelined Streaming Computation of Histogram in FPGA OpenCL

Mohammad HOSSEINABADY [1], Jose Luis NUNEZ-YANEZ
Electrical and Electronic Engineering, Bristol University, UK

Abstract. The emergence of High-Level Synthesis (HLS) techniques and tools, along with new features in high-end FPGAs such as multi-port memory interfaces, has enabled designers to utilize FPGAs not only for compute-bound but also for memory-bound tasks. This paper explains how to efficiently parallelise histogram, as a memory-bound task, utilizing the OpenCL framework running on FPGA. We have run our implementation on three high-end FPGAs including Alpha Data 7v3, Alpha Data ADM-PCIE-KU3 and Xilinx KU115. The 256 fixed-width bins histogram running on 7v3, KU3 and KU115 platforms shows 8.38, 15.29 and 38.57 Giga bin Update Per Second (GUPS), respectively. The best result, i.e., 38.57 GUPS on KU115 platform defeats the Nvidia GeForce 1060 GPU with 31.36 GUPS. In addition, it shows better performance than the one obtained in the dual socket 8-core Intel Xeon E5-2690 with 13 GUPS and 60-core Intel Xeon Phi 5110P coprocessor with 18 GUPS. The proposed implementation is not sensitive to locally invariant (LI) data sets, while the performance of GPU and CPU implementations drops with LI data. Processing locally invariant data sets shows that our FPGA implementation can be up to 91.4% and 44.9% faster than that of the GeForce 1060 and 1080 GPUs, respectively. The source codes of the designs are available at https://github.com/Hosseinabady/histogram_sdaccel.

Keywords. FPGA, High-Level Synthesis, Stream Computing, Histogram

1. Introduction

FPGAs have been used as accelerators for compute-bound applications in different fields, including image and video processing platforms, scientific applications and embedded systems. The time consuming and tedious design process based on a Hardware Description Language (HDL), was one of the main hurdles to use FPGAs in mainstream platforms. To alleviate this issue, researchers and industry have proposed high-level synthesis (HLS) techniques [1] that receive an algorithm in a high-level language such as C/C++/SystemC/OpenCL [2,3] and then transform it into a Register Transfer Level (RTL) description which can be synthesised by logic synthesis tools into FPGA configuration bitstreams.

In this paper, we explain an efficient implementation of histogram, as memory-bound task, using HLS tools. Histogram is a fundamental statistical tool in the algorithms of various fields including image processing, scientific computing, data-base analysis,

[1]Mohammad Hosseinabady, Electrical and Electronic Engineering, Bristol University, UK, BS8 1UB; E-mail:m.hosseinabady@bristol.ac.uk

data-centre analysis, profiling big data and so on. Histogram has been successfully used in image processing to enhance the image contrast [4], to be used in medical image processing, to detect objects. Histogram is also used as the pre-processing step for profiling big data in order to determine their distributions to be used in adapting data processing algorithms, appropriately [5].

Using the OpenCL framework, we explain how to parallelise the histogram utilising different features in new FPGAs including wide bus width and multiple memory ports. The novelties of this paper are as follows:

- A practical demonstration of the effectiveness of the FPGA at accelerating a typical memory-bound benchmark such as histogram
- Proposing architectural techniques to reduce the initiation interval of the pipelined loops
- Implementing the proposed techniques on three different FPGAs and evaluating the effects of locally invariant (LI) data on performance

Considering randomly distributed data, our experimental results show about 18.7% speed-up compared to the NVIDIA GeForce 1060 GPU and up to 53.3% speed up in comparison with the 60-core Intel Xeon Phi 5110P coprocessor. In addition, processing locally invariant data sets shows that our FPGA implementation can be up to 91.4% and 44.9% faster than that of the GeForce 1060 and 1080 GPUs, respectively.

The rest of this paper is organised as follows. The next section reviews the previous work. Section 3 clarifies the problem to be solved in this paper and explains our contributions. Theoretical concepts and techniques for developing an efficient implementation of the fixed-width bin histogram are discussed in Section 4. Section 5 analyses the experimental results and compares them with other state-of-the-art techniques. Finally, Section 6 concludes the paper and proposes future work.

2. Previous Work

Histogram is a well known tool that was introduced in 1895 [6], and there have been many possible versions and implementations since that time. The most advanced implementations try to run the corresponding algorithm on multi-core processors, GPUs and FPGAs to increase the performance in analysing large data streams. Jung et al. [7] proposed a parallel histogram construction on multi-core systems including 8-core Intel Xeon E5-2690 and 60-core Intel Xeon Phi 5110P. They have considered different types of histogram including fixed-width and variable-width bins. A parallel version of 256/64 bin histogram is presented by [8] for NVIDIA GPUs. In contrast to these approaches, we have implemented the fixed-width 256-bin histogram on the FPGA which shows higher performance than these methods which utilise CPUs and GPUs. A parallel pipelined array of cells was introduced by [9] for running the histogram task on small devices such as cameras. Their implementation computes two data items per clock cycle which is quite slow compared to our proposed methodology which processes up to 256 data items per clock cycle.

Researchers have studied the limitation of parallel data access in FPGA as the main bottleneck for maximizing the throughput in HLS [10,11,12]. However, this paper shows that optimising the data access should be along with minimizing the initiation interval in order to consume all the bandwidth provided by the external memory system.

3. Problem definition and Contributions

Histogram represents the distribution of data over a range of values, also known as bins. Bins can have fixed or variable width sizes. For example, the histogram of 8-bit pixels in an image can have 256 bins of size one or 64 bins of size four.

Listing 1 shows a general form of a sequential code for the histogram algorithm. It defines the *hist* array (at Line 2) which keeps the histogram and a simple loop iterates over the input data (at Line 3). The loop contains two lines of code, Line 4 calls function *findIndex* to compute the bin index corresponding to the input data and Line 5 updates the histogram invoking the *update* function.

Listing 1: Sequential histogram algorithm

```
1
2  int hist[BIN_SIZE];
3  for (int i = 0; i < DATA_LENGTH; i++) {
4    int index = findIndex(data[i]);
5    update(hist[index]);
6  }
7
```

The main contribution of this paper is proposing an efficient technique to implement this algorithm on FPGA using the OpenCL framework. In order to achieve a high performance implementation the proposed techniques try to

- Provide a pipelined stream computing
- Reduce the initiation interval of the pipeline to 1
- Utilise multiple memory ports in FPGAs

4. Fixed-Width Bin Histogram

The most common histogram uses fixed-width bins such as the 8-bit-pixel image histogram with 256 bins of size one in which each bin represents an 8-bit pixel. In the 8-bit-pixel image histogram, the *findIndex* is the identical function in which the input data value is the corresponding bin index and the *update* function increments the value of the selected bin. In other words, the *update* function is a read-modify-write operator. Therefore, an efficient implementation of the histogram on FPGA should parallelise different iterations of the *update* function at Line 5 of Listing 1. However, each iteration of the encompassing loop requires accessing the data located in the main memory as well as accessing the bins array (i.e., *hist* array in Listing 1) which has an irregular index. An efficient implementation needs to solve two problems. The first problem is transferring data from off-chip memory into the FPGA which contains a limited amount of internal memory (also known as BRAM). The second problem is running the loop iterations, with irregular access pattern, concurrently.

Reading and processing the input data elements in a streaming fashion can solve the first problem. The dataflow graph in Fig. 1 shows this idea in which a pipe connects two kernels R_Data and His. While the former kernel reads the data in streaming fashion the latter processes them using a pipeline scheme.

Figure 1. OpenCL diagram

Listing 2 shows the pseudo-code for this implementation. The *pdata* pipe declared at Line 1 is used for kernel communication. The *R_Data* kernel from Line 2 to Line 10 reads pixels and pushes them into the *pdata*. The *Hist* kernel from Line 12 to Line 22 defines the *hist* array in FPGA block RAM (BRAM) and fetches pixels from the pipe and updates the *hist* array. After updating the *hist* array for all input data, it will be transferred to the main memory (at Line 21) to be read by the host program.

The code in Listing 2 utilises loop pipelining techniques by using pragmas at Lines 4 and 16. After synthesising with the SDAccel FPGA OpenCL tool [2], as shown in Figs. 2a and 2b, the initiation interval (II) of the loops in *R_Data* and *Hist* kernels are 1 and 2, respectively. As the II_{R_Data} is one, the *R_data* kernel is synthesised to logic implementing the burst data transfer protocol. This means that, the *R_data* can potentially provide a data element in each clock cycle. However, as the II_{Hist} is two, the *Hist* kernel can consume read data every other clock cycle. The slow consumption rate of the *Hist* kernel insert stalls in the *R_Data* pipeline, as depicted in Fig. 2c. The main reason for $II_{Hist} = 2$ is the irregular data access in loop iterations which results in a read-after-write (RAW) data dependency between consecutive iterations for accessing the bins. Therefore, the *Hist* kernel would be the bottleneck for increasing the throughput of the design.

According to this discussion, the number of clock cycles to finish the histogram computation is an order of $II \times \frac{n}{m}$ in which n is the size of the input data in bytes and m is the number of bytes in each data elements read by the *R_Data* kernel. Note that the latency of the data transfer at Line 21 has been ignored in this formula as it is not dependent to the input data size. If f represents the design clock frequency, then the memory bandwidth utilisation in the ideal case without considering any overhead can be estimated by $\frac{n}{1/f(IIn/m)} = \frac{f \times m}{II}$. As in our case $II = 2$ then the execution time and ideal memory utilisation are an order of $O(2n/m)$ and $O(fm/2)$, respectively. For example, for the *int* data type which consists of four bytes and 200MHz clock frequency, the utilised memory bandwidth is around $4 * (200MHz)/(2)Gbytes/s = 400MB/s$.

To increase the design performance and memory bandwidth utilisation, two parameters should be optimised which are II and m. In the rest of this section, we will explain how to optimise these parameters.

Listing 2: Pseudo-code for FPGA OpenCL image histogram

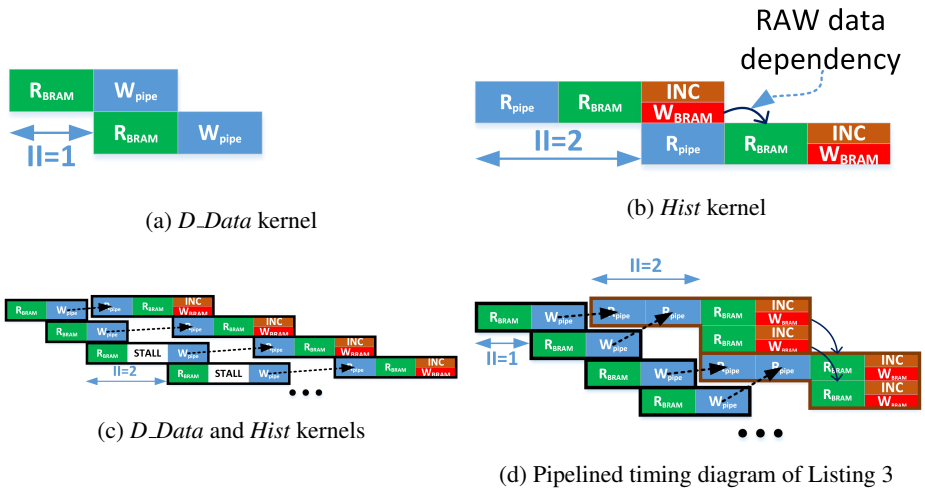
```
1  pipe  DATA_TYPE pdata;
2  kernel void R_data(__global int8* data) {
3    int global_index;
4    __attribute__((xcl_pipeline_loop))
5    for (int i = 0; i < DATA_LENGTH; i++) {
6      global_index = i;
7      DATA_TYPE d = data[global_index];
8      write_pipe_block(pdata, &d);
9    }
10 }
11 ------------------------------------------
12 kernel void Hist(__gloabl int *hist_result) {
13   int  hist[BIN_SIZE]={0};
14   DATA_TYPE d;
15   int  h;
16   __attribute__((xcl_pipeline_loop))
17   for (int i = 0; i < DATA_LENGTH; i+=1) {
18     read_pipe_block(pdata, &d);
19     hist[d]++;
20   }
21   async_work_group_copy(hist_result,  hist, BIN_SIZE, 0);
22 }
```

4.1. Reducing initiation interval

A simplified timing diagram of the pipelined loop in the *Hist* kernel for two consecutive iterations is shown in Fig. 2b. In the first cycle (i.e., R_{pipe}), a data is read from the input pipe, then using this data as bin index, the second cycle (i.e., R_{BRAM}) reads the BRAM having access to the corresponding bin. In the last cycle, the read value from BRAM is incremented (i.e., *INC*) and written back to the BRAM (i.e., W_{BRAM}). The next loop iteration cannot read the BRAM until the current iteration has modified that. This dependency dictates an initiation interval of 2 for the pipelined loop.

In order to increase the throughput, the *Hist* kernel should consume the data in the pipe with the same rate as they have been generated by the *R_Data* kernel. Our proposed technique to fulfil this constraint is to process more than one pixel in parallel in each iteration while the II remains unchanged. This can be realised by defining two separate bin arrays (as the loop $II = 2$). The modified *Hist* kernel code is shown in Listing 3 in which each iteration of the modified loop reads two pixels from the pipe and updates each histogram using one of the pixels. Although, the two tokens are read sequentially from pipe (Lines 4 and 5 of Listing 3), the bins are updated in parallel (Lines 6 and 7 of Listing 3) as there is no any dependency between them. In other words, each iteration contains two *hardware threads* to update the histogram bins as shown in Fig. 3a. Using this technique, the *Hist* kernel can consume received tokens with the same pace they are generated by the *R_Data* kernel. A simplified timing diagram in Fig. 2d depicts the timing dependency between two consecutive iterations of the loop in Listing 3.

The number of clock cycles required to finish the histogram is an order of $O(n/m)$, in which n is the size of input data and m is the number of bytes read in each memory access. In addition, $f \times m$ estimates the utilised memory bandwidth. For example, for



(a) *D_Data* kernel

(b) *Hist* kernel

(c) *D_Data* and *Hist* kernels

(d) Pipelined timing diagram of Listing 3

Figure 2. Pipelined timing diagram of kernels in Listing 2

the *int* data type and 200MHz clock frequency, the utilised memory bandwidth is around $4 * (200MHz) = 800MB/s$.

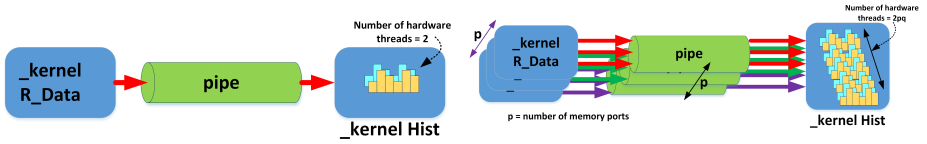
Listing 3: Modified loop of *Hist* kernel for II reduction

```
1  DATA_TYPE d_1, d_2;
2  __attribute__((xcl_pipeline_loop))
3  for (int i = 0; i < DATA_LENGTH/2; i++) {
4    read_pipe_block(pdata, &d_1);
5    read_pipe_block(pdata, &d_2);
6    hist_1[d_1]++;
7    hist_2[d_2]++;
8  }
9  __attribute__((xcl_pipeline_loop)) %
10 for (int i = 0 ; i < BIN_SIZE; i++)
11   hist[i] = hist_1[i] + hist_2[i];
12 async_work_group_copy(hist_result, hist, BIN_SIZE, 0);
```

4.2. *Increasing memory bandwidth*

There are two techniques to increase the memory bandwidth. The first is using all bus-width available on the FPGA to read multiple data elements in each memory access. This can be achieved by using OpenCL vector data types. The second technique is utilising multiple memory ports that are available in new FPGA platforms. These two techniques are explained in the sequel.

Vector data type: Using the OpenCL vector data type to read more data in each memory access increases the memory bandwidth utilisation. In this case, in each iteration the *R_Data* kernel, in one side of the pipe, generates multiple data that should be consumed by the *Hist* kernel in the other side of the pipe. To satisfy this constraint, the



(a) Reducing the design II to 1

(b) Using OpenCL vector type and multiple memory ports

Figure 3. Pipelined timing diagram of kernels in Listing 2

Table 1. 7V3 experimental results for histogram ($n = 33554432$)

optimization	f	FPGA exe. time (msec)			GUPS exe. time (msec)	FPGA Resource Utilisation		
		R_Data	*Hist*	total		FF	LUT	BRAM
none[1]	200	335.685	335.637	335.97	0.09987	2101	3005	5
II[2]	200	167.879	167.832	168.146	0.1996	2399	3273	12
II + *vec*[3]	200	3.646	3.595	4.003	8.38	25496	27830	290

[1] No-optimisation [2] Reducing *II* [3] Using vector data type

Hist kernel utilises more hardware threads (i.e., parallel iterations) to update bins, similar to the techniques explained for reducing II in Subsection 4.1.

In this case, for *int16* OpenCL vector data type m is 4×16. If $200MHz$ clock frequency is considered for the design, then the utilised memory bandwidth is around $4 * 16 * (200MHz) = 12.800GB/s$.

Multiple memory ports New FPGAs provide multiple memory ports to have access to the data. If the data is located in different memory banks then utilising multiple ports directly increases the memory bandwidth utilisation. Fig. 3b depicts the idea of using multiple ports along with the OpenCL vector data type. For each memory port, we have defined an *R_Data* kernel which pushes the read data into a dedicated pipe. Then the *Hist* kernel reads the data streams from pipes and updates the separate bins in parallel.

In this case, if the number of ports denoted by p is 4 and using the *int16* OpenCL vector data type, then $m = 4 \times 16 \times 4$. In addition, if $200MHz$ clock frequency is considered for the design, then the utilised memory bandwidth is around $4 * 4 * 16 * (200MHz) = 51.200GB/s$.

5. Experimental Results

This section explains the results of implementing the proposed techniques and compares them with the results of using CPUs and GPUs.

5.1. FPGA implementation

Figs. 1, 2, and 3, shows the results of implementing and running the proposed methods on three high-end Xilinx FPGA including ADM-PCIE-7V3 [13], ADM-PCIE-KU3 [14] and Xilinx acceleration KU115 4DDR expanded partial reconfiguration platform [15], respectively. In addition, we use the Xilinx tool sets to compile the OpenCL codes. These results are obtained by running the histogram designs on the FPGA board provided by NIMBIX cloud [16]. The source code of these implementations can be found at [17].

Table 2. KU3d experimental results for histogram ($n = 33554432$)

optimization	f	FPGA exe. time (msec)			GUPS exe. time (msec)	FPGA Resource Utilisation		
		R_Data	*Hist*	total		FF	LUT	BRAM
none[1]	200	335.677	335.619	335.89	1.0	2907	2987	5
II[2]	200	167.907	167.847	168.155	1.995	2303	3273	14
$II + vec$[3]	200	2.994	3.086	3.403	9.86	18548	27830	290
$II + vec + mp$[4]	200	1.55	1.955	2.195	15.29	38414	65197	602

[1] No-optimisation [2] Reducing *II* [3] Using vector data type [4] Using multiple memory ports

Table 3. KU115 experimental results for histogram ($n = 33554432$)

optimization	f	FPGA exe. time (msec)			GUPS exe. time (msec)	FPGA Resource Utilisation		
		R_Data	*Hist*	total		FF	LUT	BRAM
none	260	258.220	258.167	258.50	0.129	2097	2987	5
II	300	111.963	111.906	112.189	0.299	2443	03273	12
II+vec	234	2.298	2.380	2.66	12.61	25496	27830	290
II+vec+mp	190	0.71	0.75	0.87	38.57	293387	197592	1034

[1] No-optimisation [2] Reducing *II* [3] Increasing f [4] Using vector data type

As it can be seen increasing the bus-width and the number of memory ports directly increases the performance, however the logic synthesis may struggle to satisfy the timing constraints. This is the reason of low operating frequency in Table. 3 for using wide bus-width and multiple memory ports.

5.2. Comparison

Considering the three FPGA implementations and their results explained in the previous subsection, here we compare the obtained performance with other platforms and state-of-the-art techniques proposed in previous research. To compare with GPUs, we have considered two Nvidia GPUs: Quadro K600 and GeForce GTX 1060 3GB. Whereas, the former is classified as a typical graphic processor the latter is a GPU suitable for GPGPU computing. The OpenCL histogram implemented for GPU is based on the code [8] optimised for NVIDIA GPUs. To compare with multi-core CPUs, we have considered four different CPUs including the 8-core Intel iCore 7, 8-core Intel Xeon E5-2690 [7], 60-core Intel Xeon Phi [7]. Fig. 4 compares the performance of the 256 fixed-width bin histogram on different platforms considering the uniform randomly distribution data. According to the graph, the FPGA implementation accessing four memory ports represents about 53.3% and 18.7% performance improvement compared to that of the fastest CPU and GPU, respectively.

5.3. Locally invariant (LI) data

The performance of a GPU implementation can drop if some of the hardware threads running in parallel (also called *warp* by NVIDIA and *waveform* by AMD) update the same location in the memory which can cause data access conflicts. In this case, the update operations should be serialised which diminish the performance of the GPU. This may happen in the histogram task in the event of locally invariant (LI) data in which adjacent data elements update the same bin in the histogram. For example, images with a uniform background or taken in the night with a dark background suffer from LI data. Fig. 5 shows the impact of the LI data on the GPU implementation by the fastest FPGA

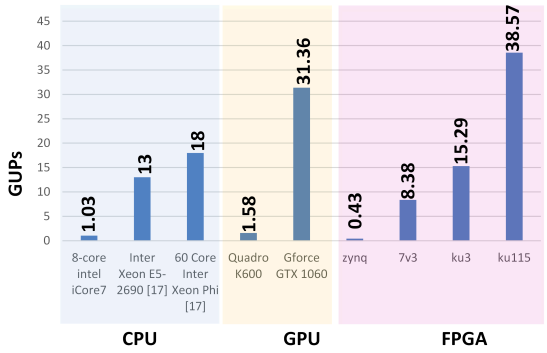


Figure 4. FPGA/GPU/CPU performance comparison

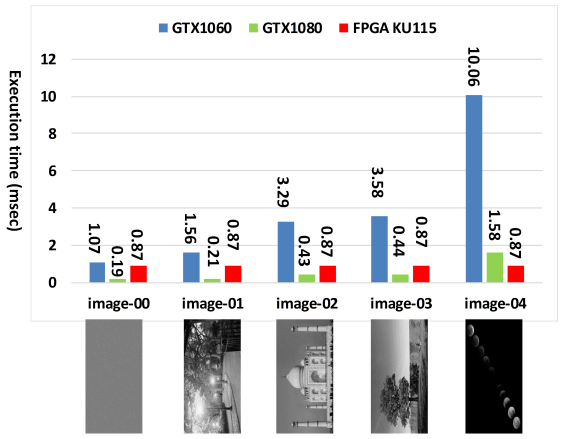


Figure 5. LI data impact

implementation, which is the fully optimised design running on KU115, with the GTX 1060 and GTX 1080 GPUs which are among the high-end GPUs available. The first image, denoted by *image-00*, contains randomly generated data, in which the probability of LI is very low, and shows a high performance on GTX1080 GPU. The FPGA implementation of histogram shows a high performance for the last image that contains a black background with high probability of LI. Note that, the FPGA implementation is not sensitive to LI as it receives the data serially and uses parallel pipelined hardware threads to update the bins that are dedicated to each pipeline, without any conflicts. The results of the last image shows that our FPGA implementation can be up to 91.4% and 44.9% faster than that of the GeForce 1060 and 1080 GPUs, respectively.

6. Conclusion and Future Work

This paper has explained how a fixed-width bin histogram can be parallelised efficiently on an FPGA OpenCL framework. The proposed parallelised implementation utilises different resources in the FPGA such as multi-port memory access and wide-bus width to

provide a fast implementation. This research can be extended in two main directions. As the resources in the FPGA are limited, the first direction of the research can study the behaviour of large fixed-width bins histograms on the FPGA-OpenCL framework. The second line of research can study the histogram implementation's power and energy requirements in FPGAs as they potentially consume less power that GPUs and CPUs.

References

[1] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[2] "SDAccel development environment," Xilinx All Programmable, Accessible 2017. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[3] D. Olds, "Are FPGAs the answer to the compute gap?," Intel, insideHPC, Tech. Rep., 2016.

[4] G. Thomas, D. Flores-Tapia, and S. Pistorius, "Histogram specification: A fast and flexible method to process digital images," *IEEE Transactions on Instrumentation and Measurement*, vol. 60, no. 5, pp. 1565–1578, May 2011.

[5] Z. Istvan, L. Woods, and G. Alonso, "Histograms as a side effect of data movement for big data," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 1567–1578. [Online]. Available: http://doi.acm.org/10.1145/2588555.2612174

[6] K. Pearson, "Contributions to the mathematical theory of evolution. II. skew variation in homogeneous material," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 186, pp. 343–414, 1895. [Online]. Available: http://rsta.royalsocietypublishing.org/content/186/343

[7] W. Jung, J. Park, and J. Lee, "Versatile and scalable parallel histogram construction," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, pp. 127–138.

[8] V. Podlozhnyuk, "Histogram calculation in cuda," NVIDIA Corporation, 2007.

[9] J. Cadenas, R. S. Sherratt, P. Huerta, W. C. Kao, and G. M. Megson, "C-slow retimed parallel histogram architectures for consumer imaging devices," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 2, pp. 291–295, May 2013.

[10] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in c-based high-level synthesis," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 166–173.

[11] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "System-level optimization of accelerator local memory for heterogeneous systems-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 435–448, March 2017.

[12] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 43:1–43:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062208

[13] "Alpha data- high performance reconfigurable computing," accessible 2017. [Online]. Available: http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3

[14] "Alpha data high performance reconfigurable computing," 2017. [Online]. Available: http://www.alpha-data.com/dcp/products.php?product=adm-pcie-ku3

[15] "SDAccel platform reference design user guide: Developer board for acceleration with KU115," Xilinx All Programmable, Tech. Rep., 2016.

[16] NIMBIX, "Nimbix cloud computig platform." [Online]. Available: https://platform.jarvice.com/landing

[17] M. Hosseinabady, "Histogram SDAccel code," 2017. [Online]. Available: https://github.com/Hosseinabady/histogram_sdaccel